Introduction à la gestion mémoire

Olivier Zendra

Introduction à la gestion mémoire

Olivier Zendra
Chargé de Recherche
INRIA-Lorraine / LORIA

olivier.zendra@loria.fr
http://www.loria.fr/~zendra

Plan du cours

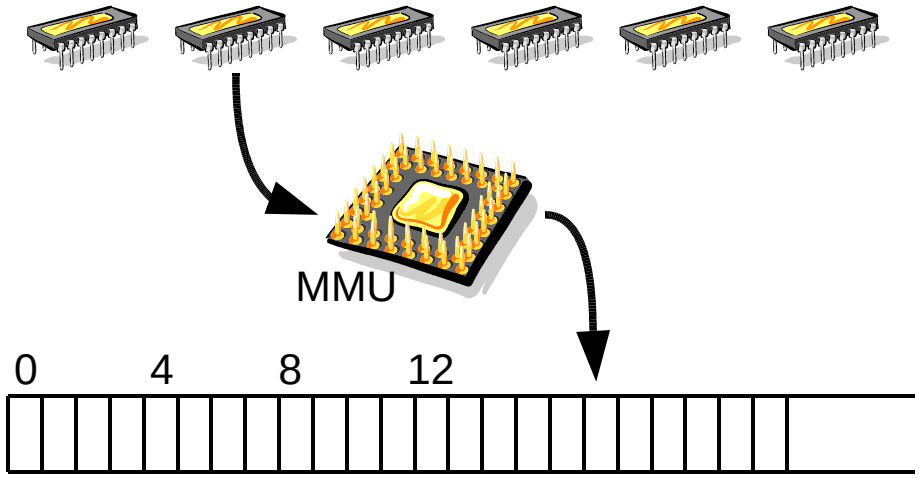
- 1- Gestion mémoire: les bases
- 2- Gestion mémoire et temps réel: éléments
- 3- Gestion mémoire et systèmes embarqués: éléments
  - Basse consommation

1- Bases de la gestion mémoire

- Concepts de base: tas, pile
- Gestion manuelle / automatique
- Algorithmes classiques:
  - comptage de références
  - marquage-balayage
  - copie / compactage

Concepts de base en gestion mémoire

- Mémoire: des puces (matériel)
- Vue par le système (OS/application) via des adresses (logiciel)



Concepts de base en gestion mémoire

- Manipulation des adresses à la main (ASM):

```
MOV 47,#0xFBBFC
MOV 74,#0xFBFBC
ADD 3,#0xFBFBC
INC #0xFBBFC
SUB #0XFBBFC,#0xFBFBC
```

  - Peu clair...

Concepts de base en gestion mémoire

- Manipulation des adresses à la main (ASM)
  - Permet de mettre des données en un lieu précis de la mémoire
  - Compliqué si on veut faire cohabiter plusieurs applications
  - Lisibilité et maintenabilité pitoyables

Concepts de base en gestion mémoire

- Manipulation symbolique explicite des adresses: variables et pointeurs

```
int *a = 0xFBBFC
int *b = 0xFBFBC
*a = 47
*b = 74
*b = *b+3
*a = *a+1
*b = *b-*a
```

  - C'est mieux: plus haut niveau, plus clair

Concepts de base en gestion mémoire

- Manipulation symbolique implicite des adresses: variables et références

```
int a = 47
int b = 74
b += 3
a++
b -= a
```

Concepts de base en gestion mémoire

- Manipulation symbolique implicite des adresses: variables et références
  - Encore plus haut niveau, plus clair
  - Localisation (adresse) masquée (par le système qui gère la mémoire)
    - Facile d'avoir plusieurs programmes (« multi-tâche »)

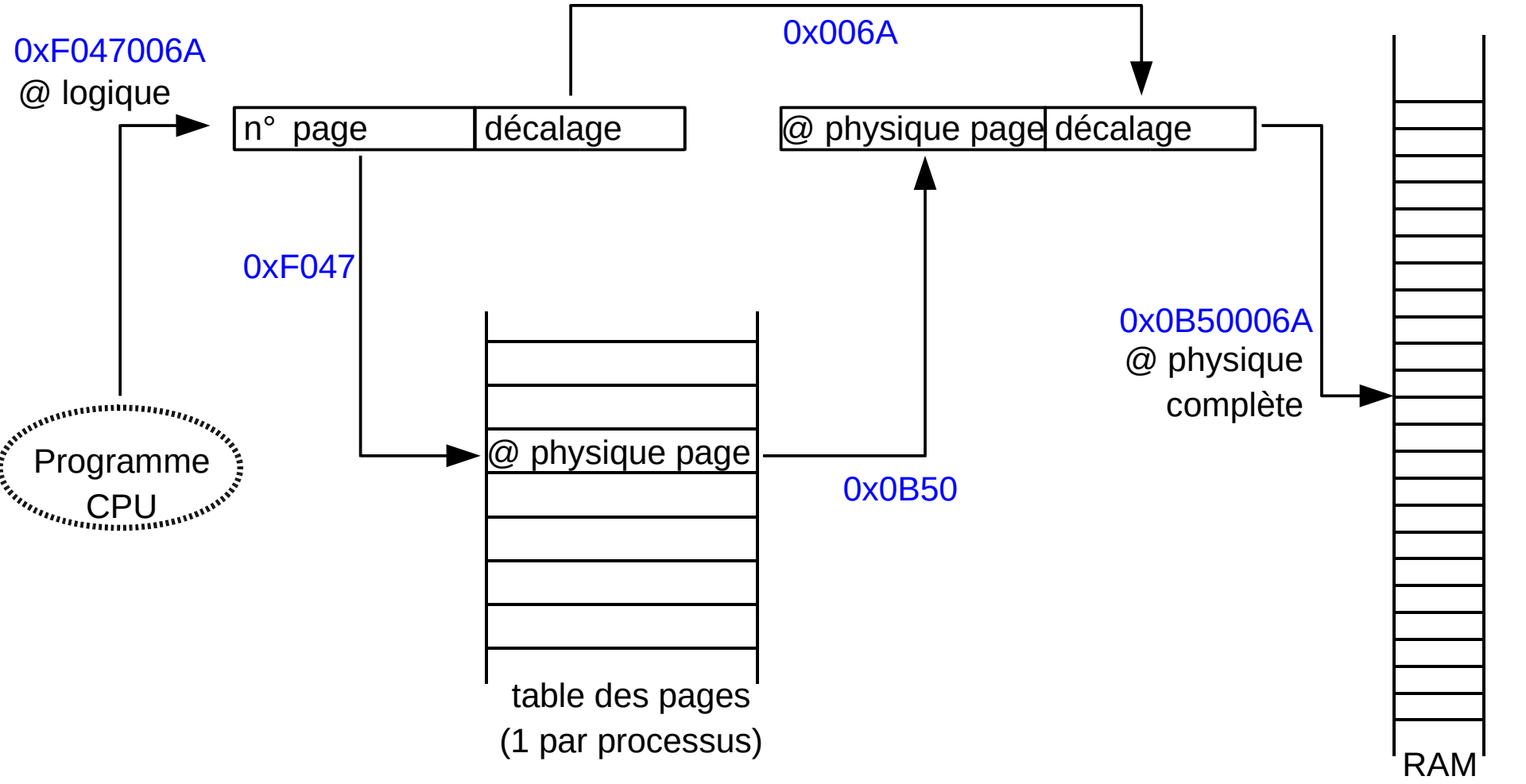
Concepts de base en gestion mémoire

- Le contrôleur mémoire (MMU) montre à l'OS une mémoire matérielle continue alors que puces discontinues
- Le système (d'exploitation) montre au programme une mémoire virtuelle
  - La pagination est masquée
  - Chaque programme se croit seul (simplifié)

Mémoire virtuelle

- Mise en correspondance mémoire virtuelle / mémoire réelle
  - traduction d'adresses
  - mémoire virtuelle >> mémoire réelle
  - mémoire virtuelle vue comme une seule zone (continue) alors que mémoire réelle discontinue (en pages)

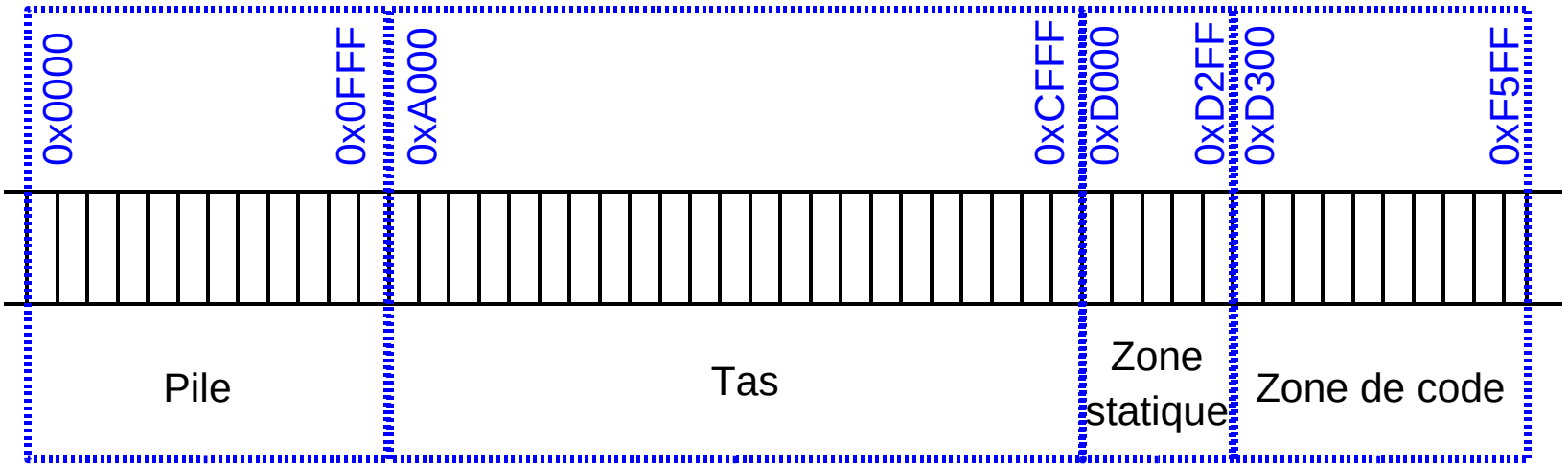
Mémoire virtuelle



Structuration mémoire

- Mémoire d'un programme découpée en plusieurs zones:
  - zone(s) statique(s) (RAM, voire ROM)
  - zone(s) de code (lecture seule)
  - zone(s) de données (lecture-écriture)
    - pile
    - tas

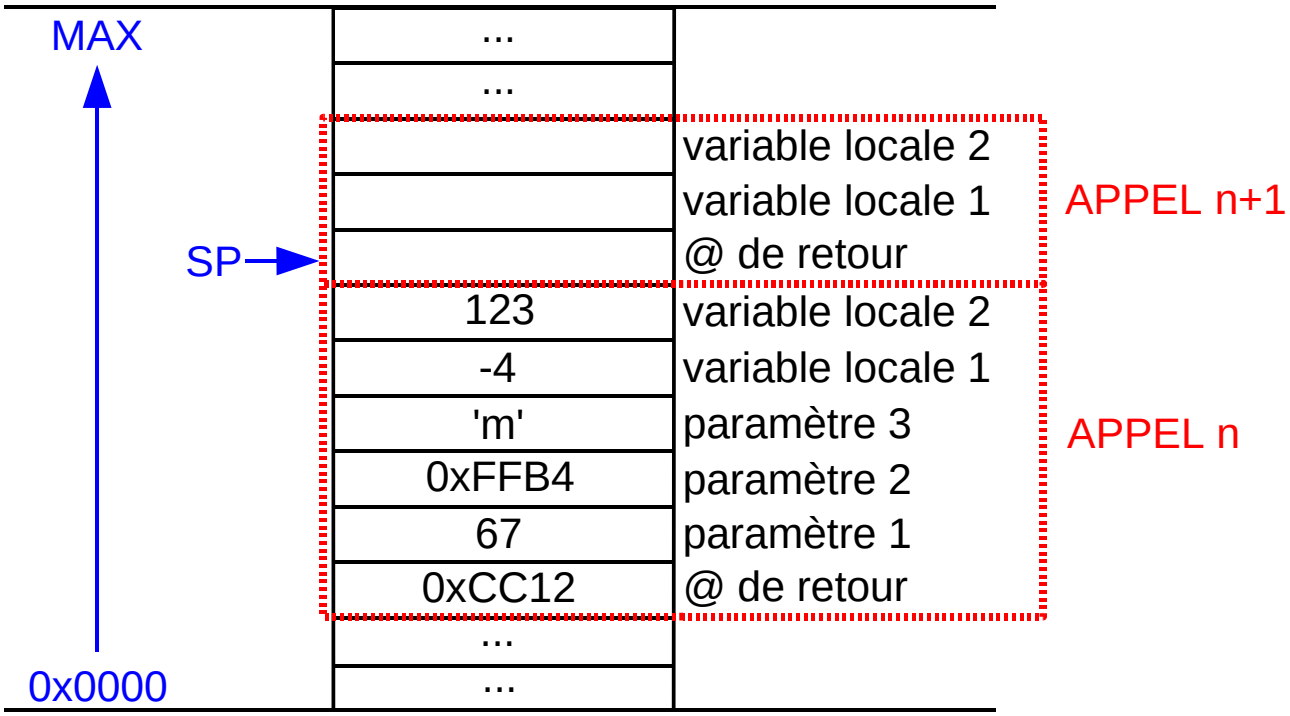
Structuration mémoire: exemple



La pile

- Zone « dans laquelle s'exécute le programme »
  - paramètres, variables locale, adresse et valeur de retour de fonction / routine / méthode
  - croît à chaque appel, décroît à chaque retour
- Automatique: géré par l'environnement d'exécution (*runtime*)
- +/- invisible du programme(ur)

La pile



NB: Ici, pile croissante avec adresses croissantes.
En pratique, la pile croît souvent vers 0x0000.

Le tas

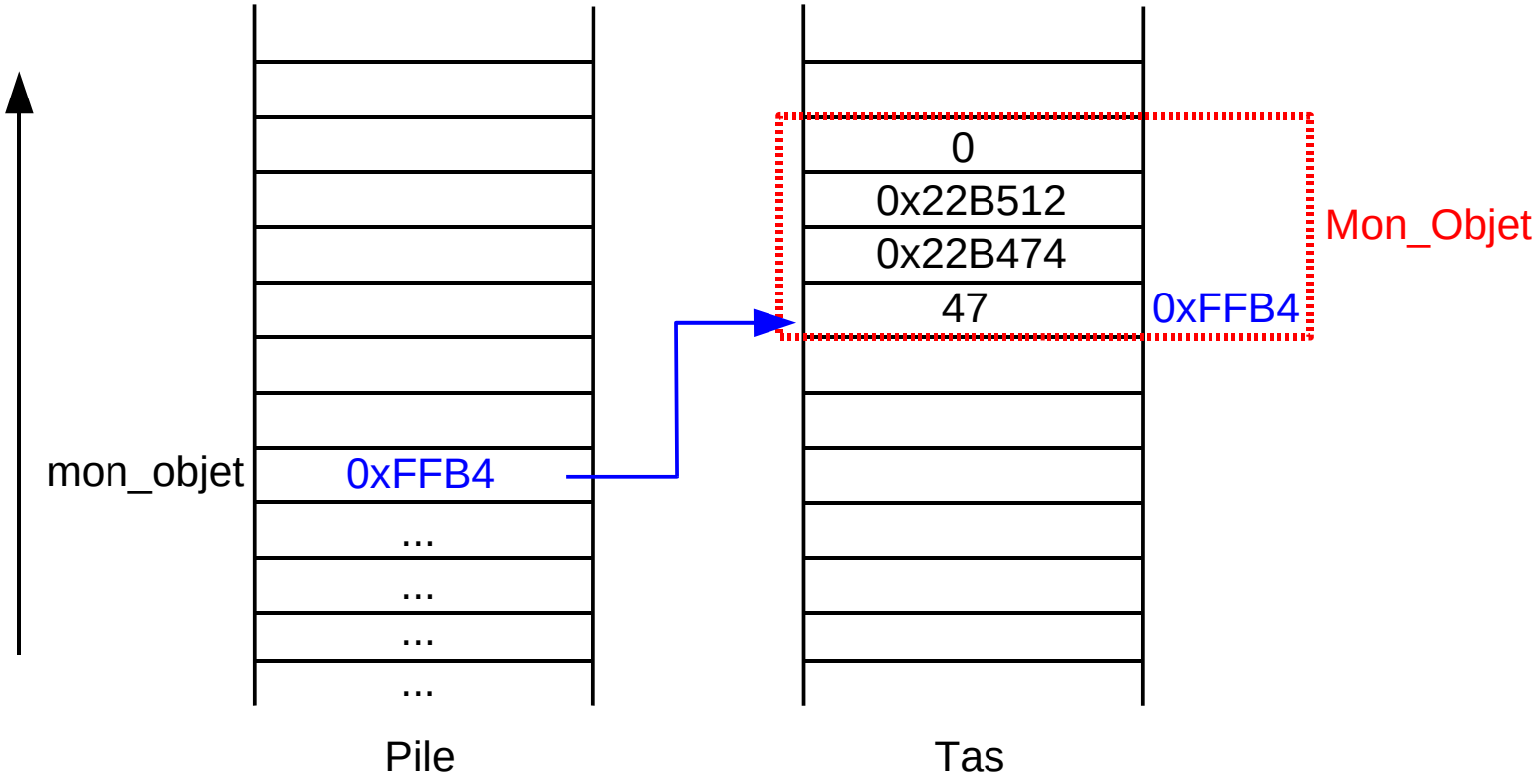
- Zone où le programme(ur) alloue toutes ses données qui ne sont pas en pile
- Allocation explicite: malloc C

```
MonObjet*mon_objet=(MonObjet*)malloc(sizeof(MonObjet));
```

- Allocation « implicite »: new en Java, C++...

```
MonObjet mon_objet = new MonObjet();
```

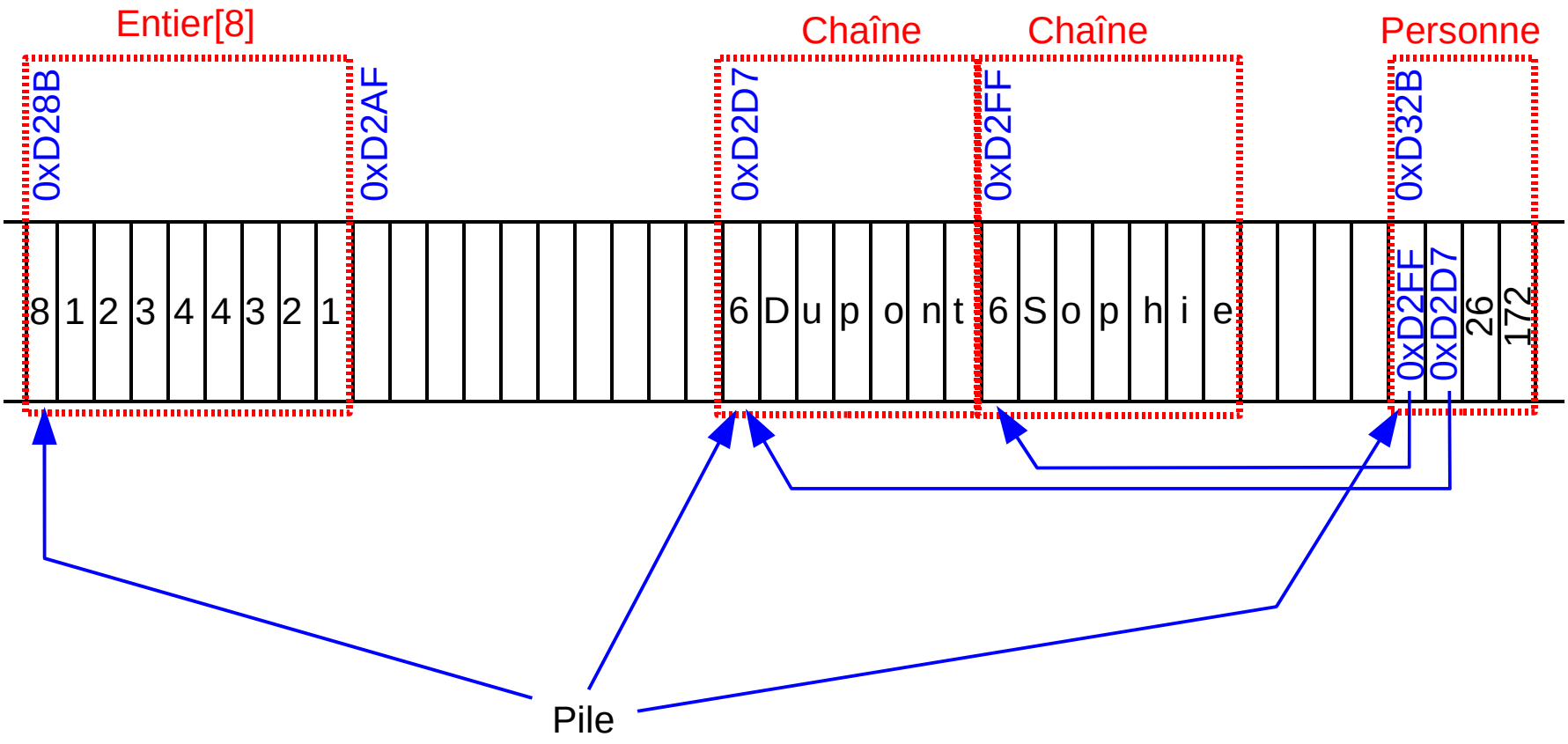
Le tas


mon_objet
0xFFB4
...
...
...
...
Pile
0
0x22B512
0x22B474
47
0xFFB4
Mon_Objet
Tas

Le tas

- Zone « désordonnée », contrairement à la pile (d'où les noms...)
- La gestion mémoire concerne principalement le tas

Le tas


Entier[8]
Chaîne
Chaîne
Personne
0xD28B
0xD2AF
0xD2D7
0xD2FF
0xD32B
8 1 2 3 4 4 3 2 1
6 D u p o n t
6 S o p h i e
0xD2FF
0xD2D7
26
172
Pile

Gestion mémoire manuelle

- Allocations et désallocations (libérations) explicites (malloc/free):

```
void les_20_premiers_premiers(){
    int premiers[]=malloc(20*sizeof(int));
    // variable premiers en pile, zone en tas
    for (int i=0;i<20;i++)
          premiers[i]=calculer_premier(i);
    afficher_tableau_entiers(20,premiers);
    free(premiers); // si pas là, fuite dans tas
}
```

Gestion mémoire manuelle

- Problème: risques d'oublis
  - mémoire non libérée (gaspillage)
  - utilisation d'une donnée déjà libérée (erreur !)
  - extrêmement difficiles à détecter et corriger

Gestion mémoire automatique

- Un système remplace le développeur en allouant et surtout désallouant automatiquement
  - impossible d'oublier de désallouer (mais possible de continuer de référencer à tort)
  - plus difficile (si pointeurs) voire impossible (si références) d'utiliser une donnée déjà libérée

Gestion mémoire automatique

- Exemple:

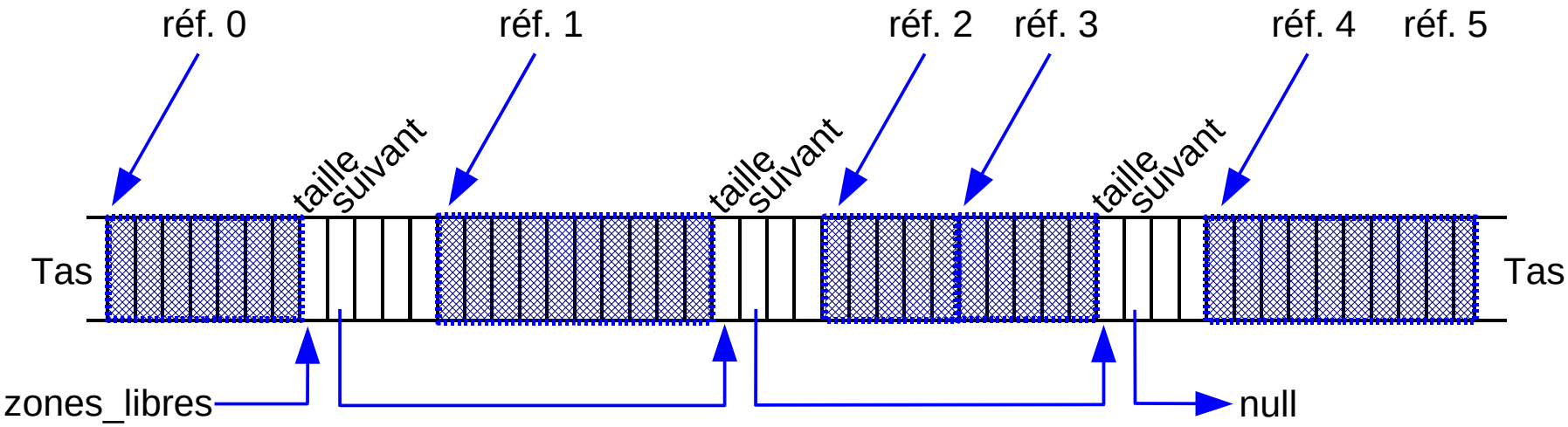
```
void les_20_premiers_premiers(){
    int premiers[] = new int[20];
    // variable premiers en pile, zone en tas
    for (int i=0;i<20;i++)
          premiers[i]=calculer_premier(i);
    afficher_tableau_entiers(20,premiers);
} // ici, premiers est dépilé, donc la zone
  // en tas n'est plus référencée, elle peut
  // être automatiquement recyclée/libérée
```

Gestion mémoire automatique

- Système automatique de gestion mémoire = ramasse-miettes
- Doit pouvoir garder (ou retrouver) les données/objets encore actifs, et du coup être capable de libérer les autres
- Connaît les zones mémoire libres (allouables) et celles qui sont occupées

Gestion mémoire automatique

- Maintient une liste des zones libres:



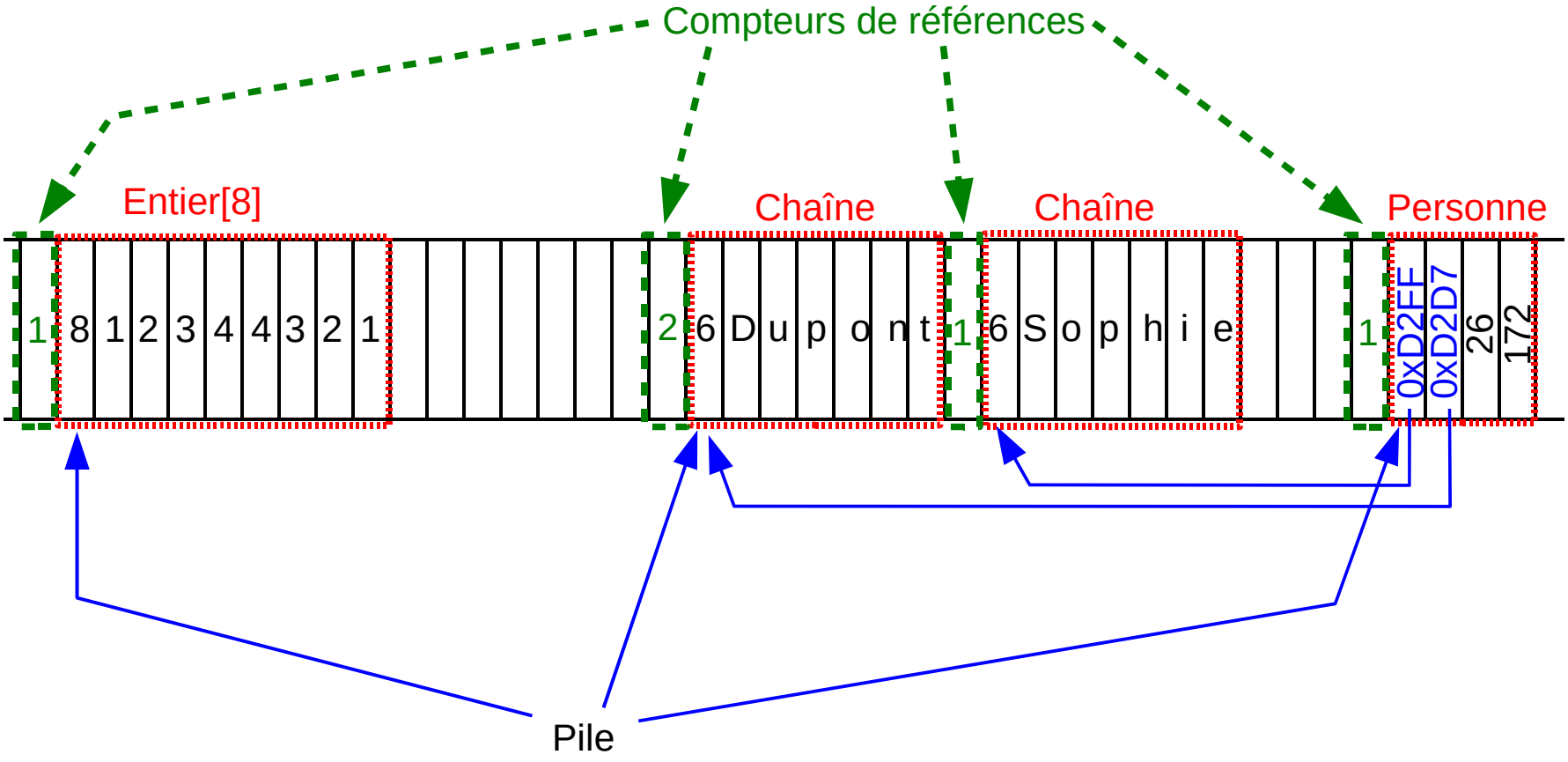
Algorithmes classiques de gestion mémoire automatique

- Comptage de références
- Marquage-balayage
- Copie-compactage

Comptage de référence

- A chaque objet (ou donnée, structure, zone mémoire) alloué est associé un compteur entier indiquant le nombre de références sur cet objet

Comptage de référence



Comptage de référence

- A chaque affectation, on met à jour les compteurs concernés:

```
a = new X();    // nb_réf(X1)=1
b = a;          // nb_réf(X1)=2
a = new X();    // nb_réf(X1)=1; nb_réf(X2)=1
b = null;       // nb_réf(X1)=0: X1 libérable
```

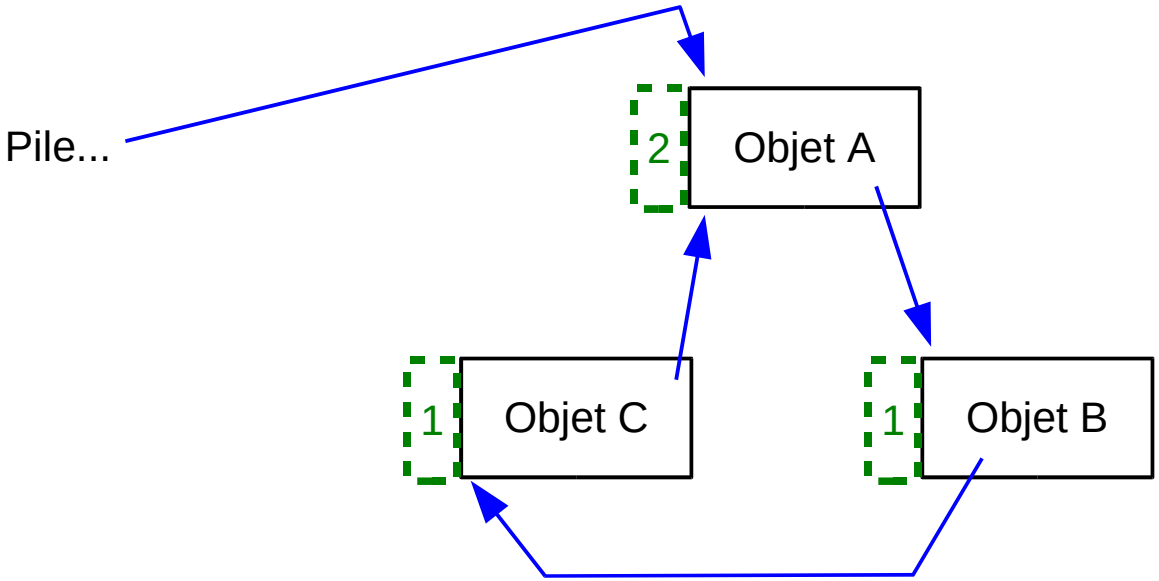
- Libération peut être immédiate (+ simple) ou différée

Comptage de référence

- Libérer un objet X1:
  - récupérer la mémoire de X1 (remise en liste libre)
  - diminuer les compteurs de références des objets pointés par X1
    - libérations en cascade possibles (peut prendre du temps)

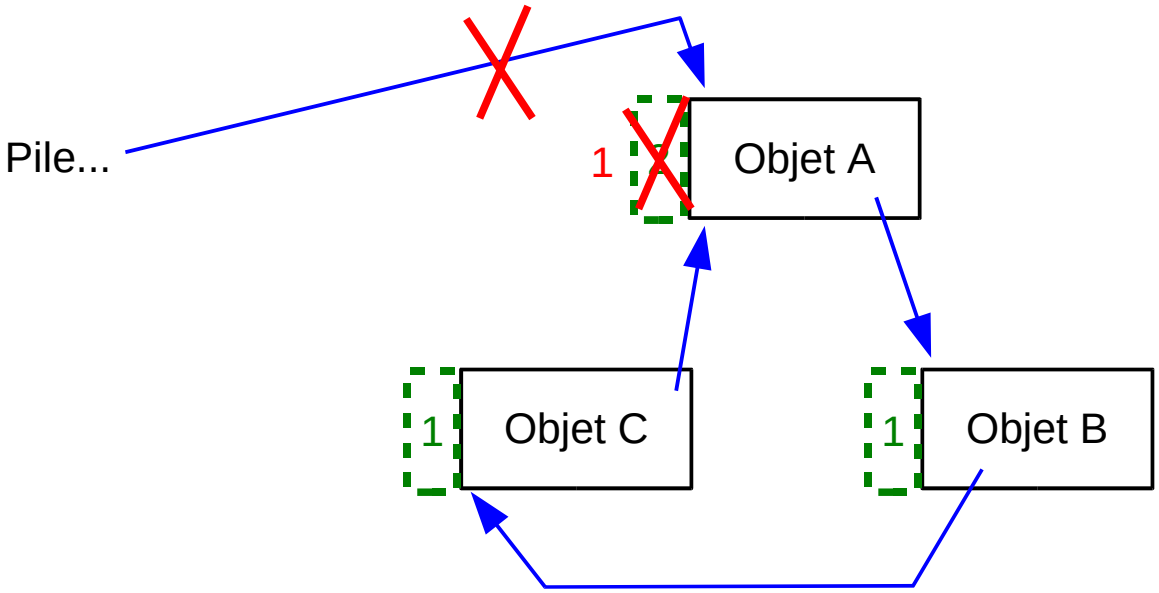
Comptage de référence

- Le problème des cycles



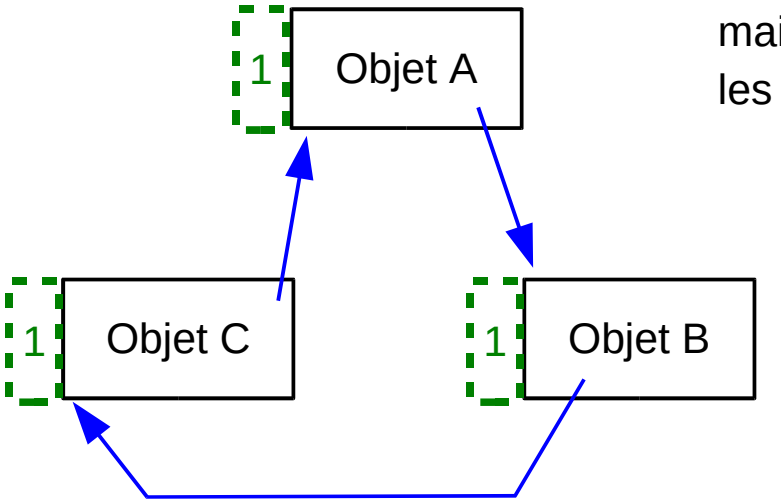
Comptage de référence

- Le problème des cycles



Comptage de référence

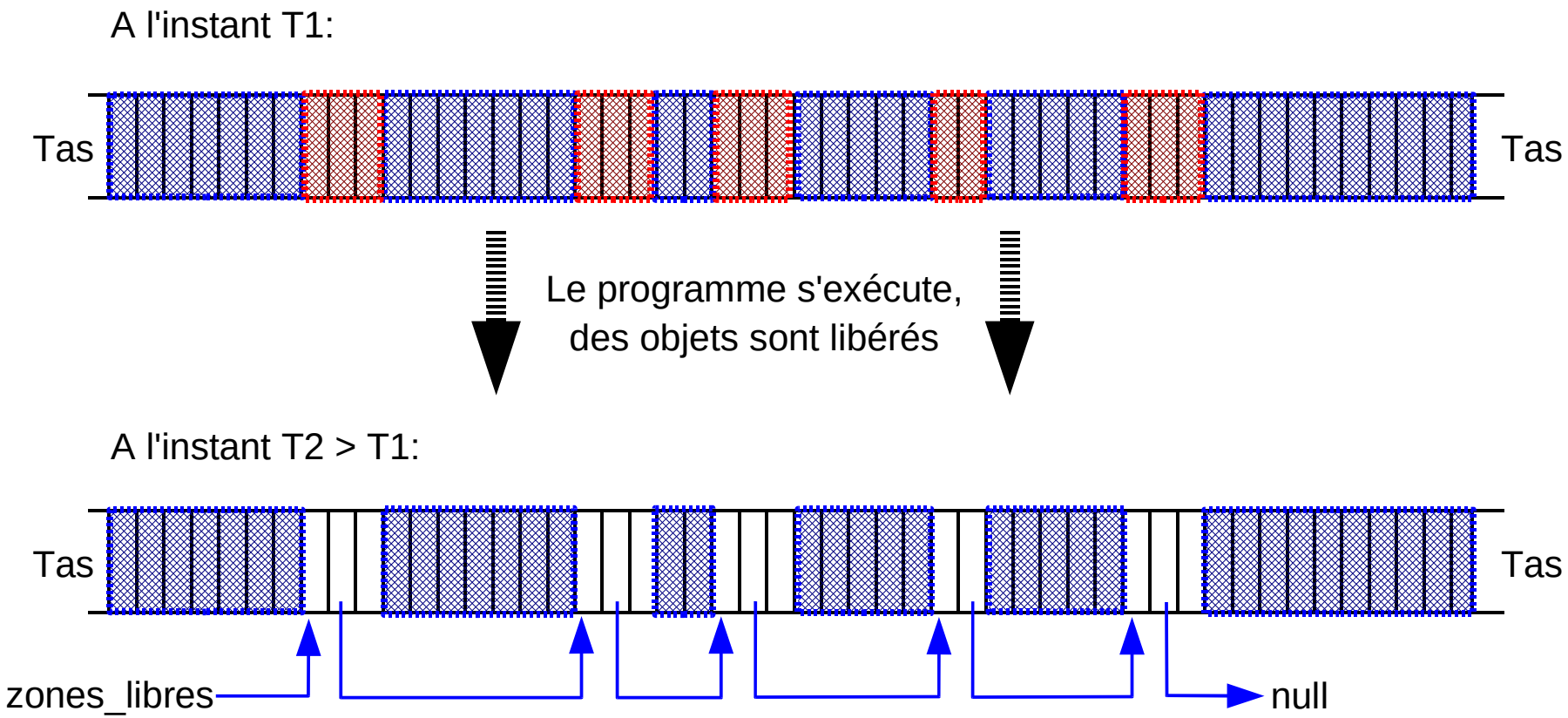
- Cycles non détectés
  - Besoin en plus d'un système de détection de cycles



Le cycle A/B/C n'est plus référencé, mais ses compteurs sont >0, donc les objets ne sont pas collectés...

Comptage de référence

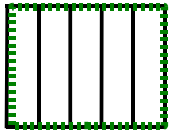
- Problème de la fragmentation



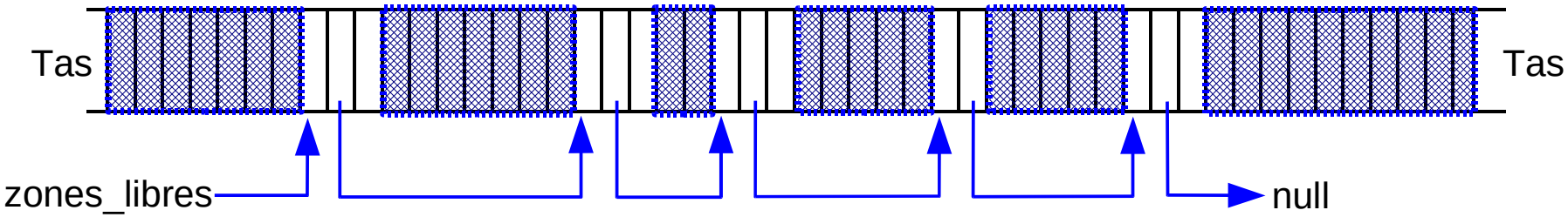
Comptage de référence

- Fragmentation possible

A l'instant T2, on veut allouer 5:



Mais pas de zone mémoire assez grande disponible:



Pourtant il y a de la mémoire libre (14 en tout): c'est la fragmentation.

Comptage de référence: bilan

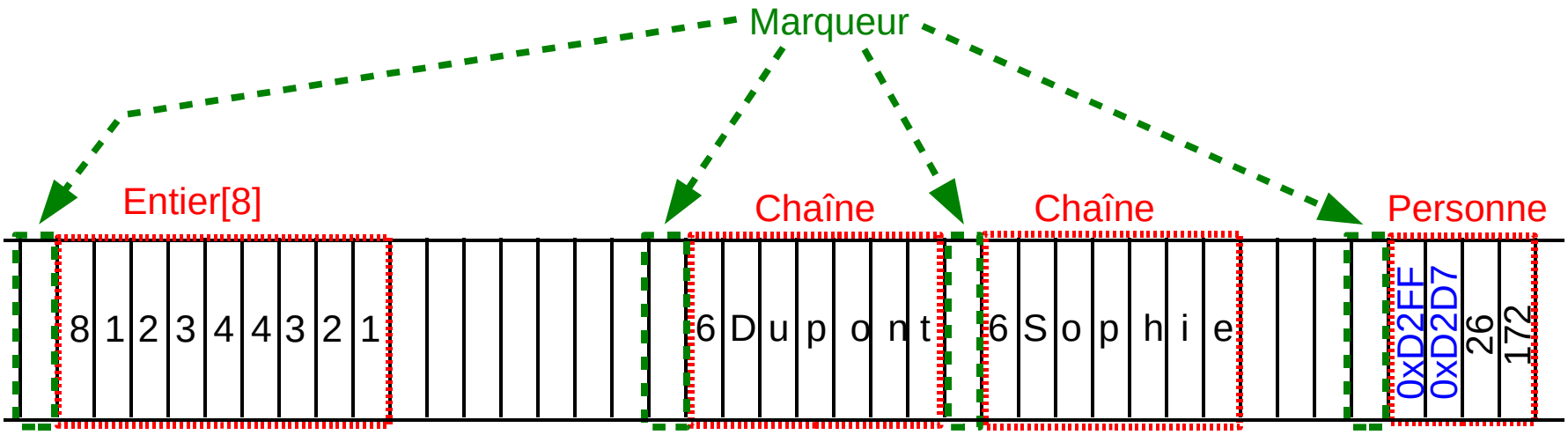
- Simple
- Exécution répartie le long de celle du prog.
- Coûteux au total: MAJ de compteur(s) à chaque affectation
- Pas de gros délai si objets libérés un par un. Délais si cascades...
- Problème des cycles
- Fragmentation possible

Marquage-balayage

- Le ramasse-miettes se déclenche par intermittence
  - Exécution du ramasse-miettes arrête le programme temporairement
- Lorsque ramasse-miettes se déclenche:
  - phase de marquage: trouver les vivants
  - phase de balayage: recycler les morts

Marquage-balayage

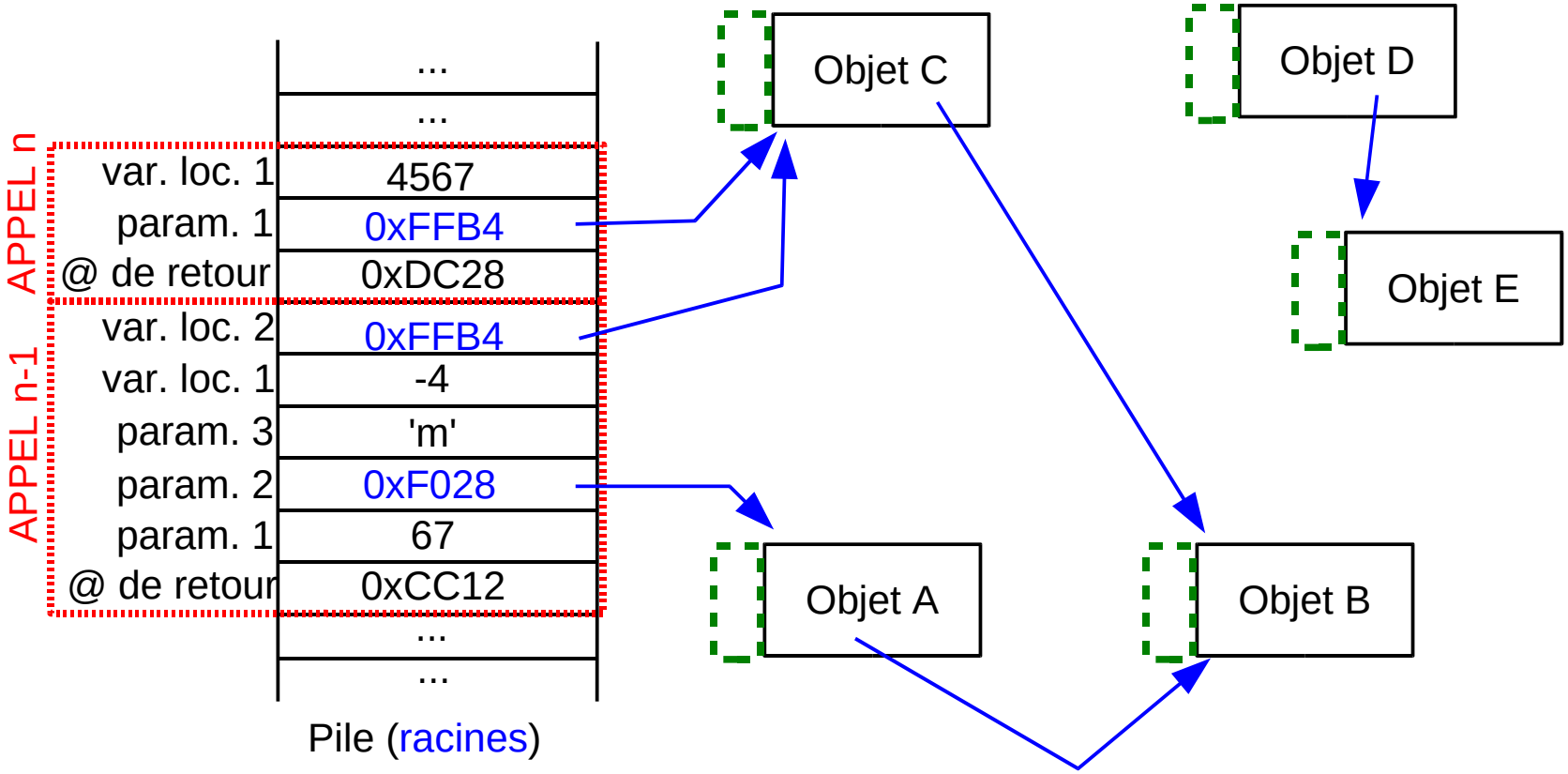
- A chaque objet alloué est associé un marqueur (ou drapeau, ou *mark flag*)



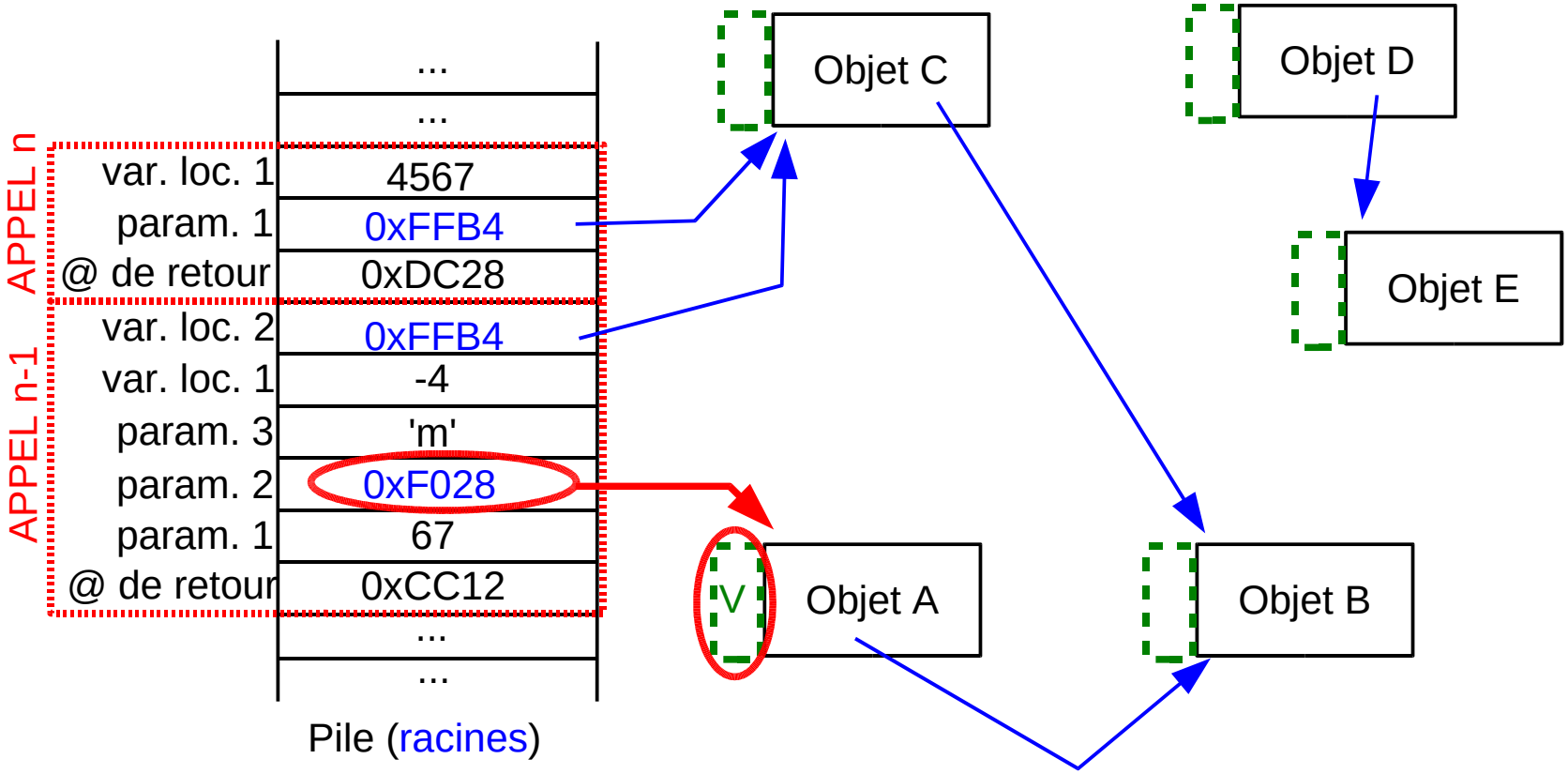
Algorithme de marquage

- Partir des racines (piles, zone statique) du graphes d'objets
- Pour chaque objet rencontré
  - s'il est déjà marqué, rien à faire
  - sinon le marquer et propager l'algorithme sur tous les objets qu'il référence
- Quand le marquage se termine, on a tous les actifs. Les autres sont morts.

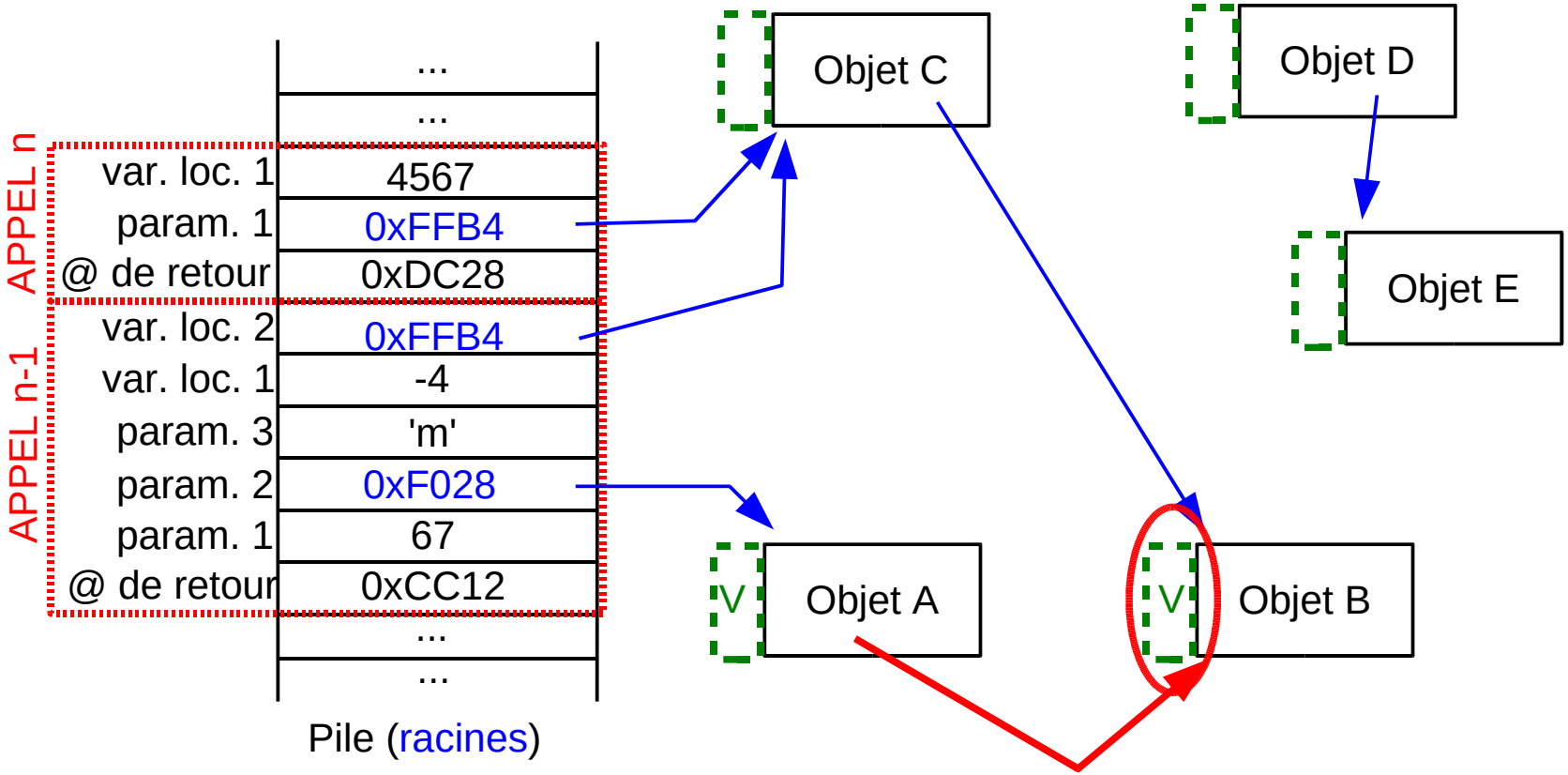
Algorithme de marquage



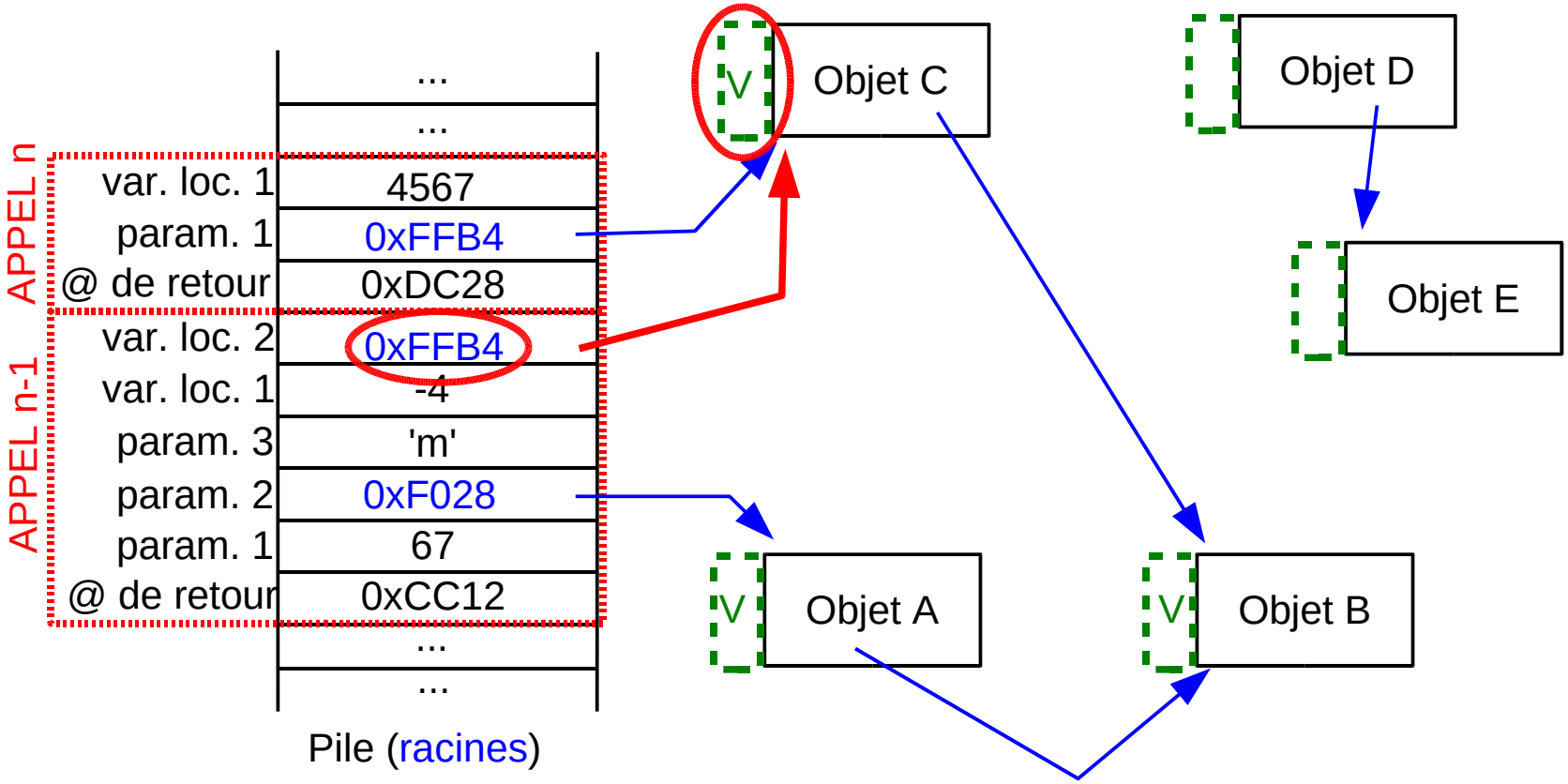
Algorithme de marquage



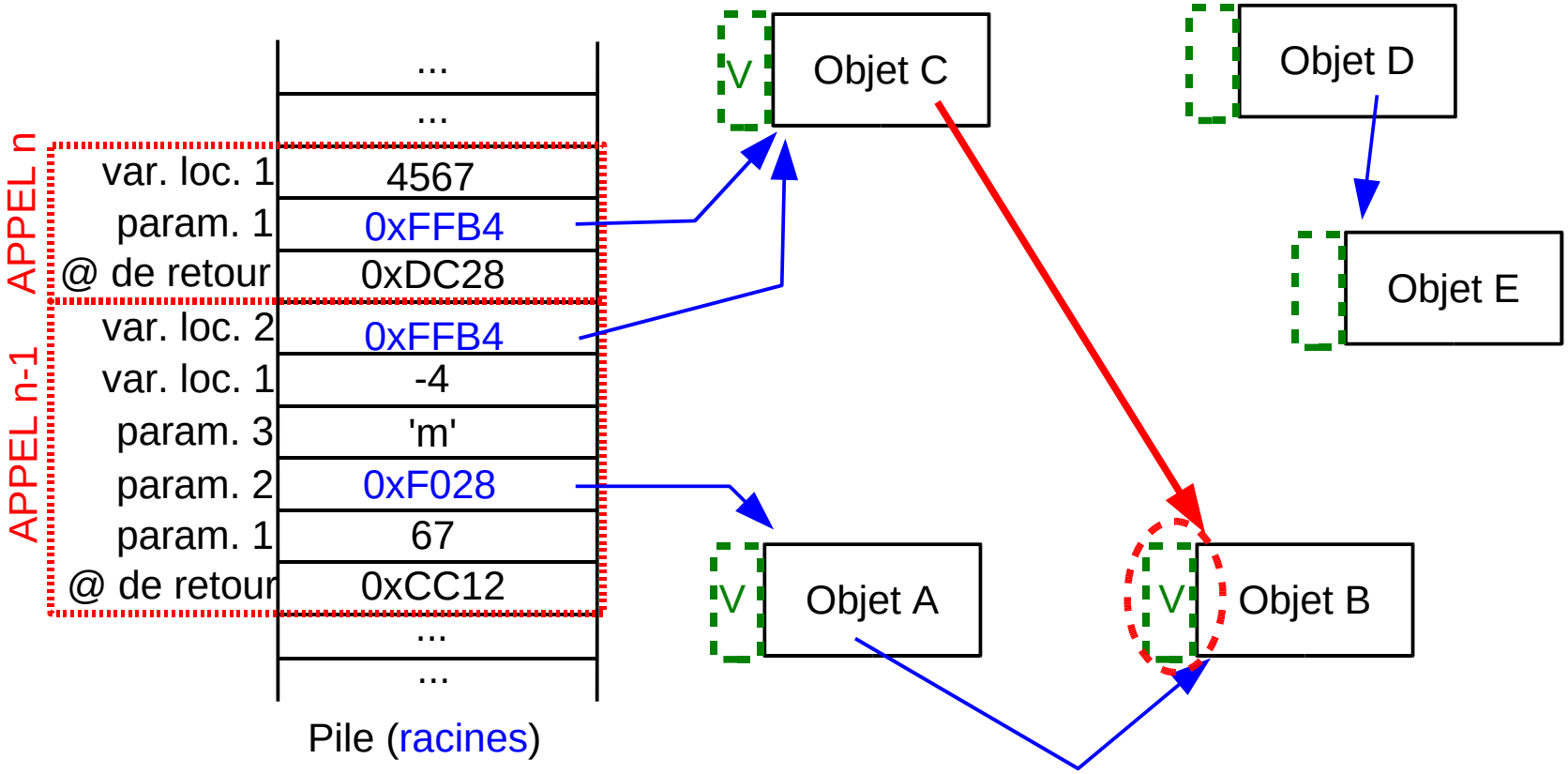
Algorithme de marquage



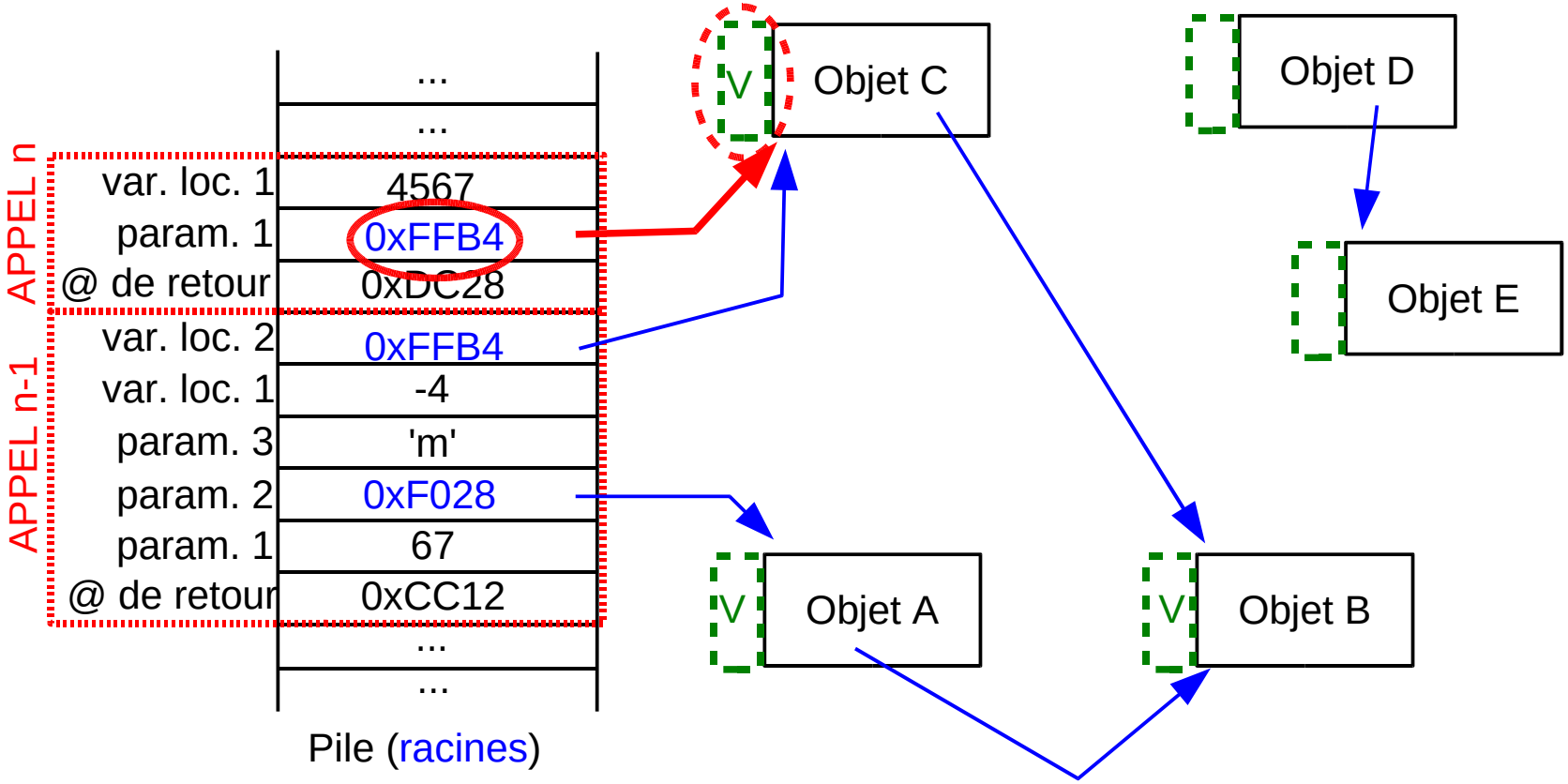
Algorithme de marquage



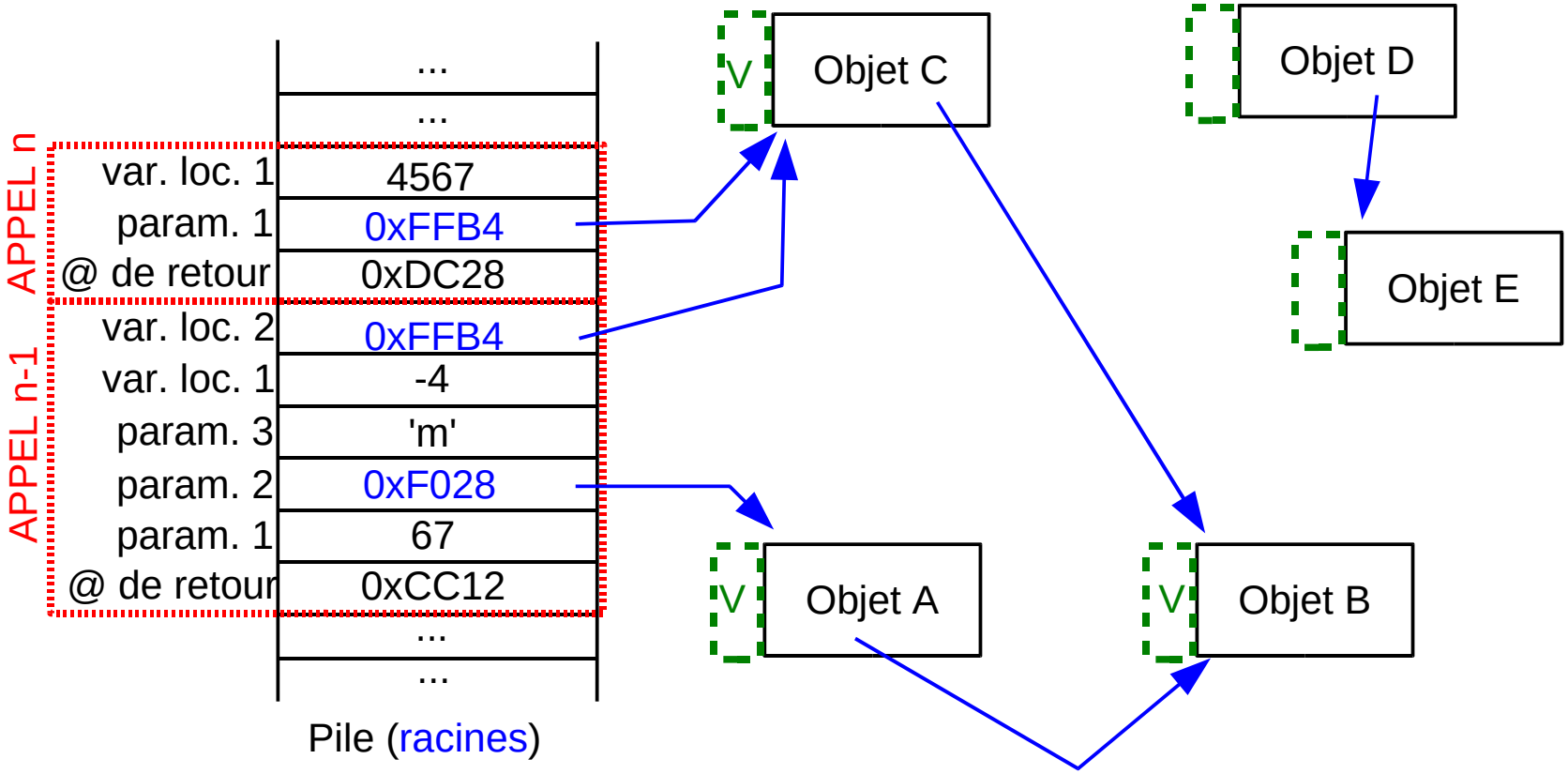
Algorithme de marquage



Algorithme de marquage



Algorithme de marquage

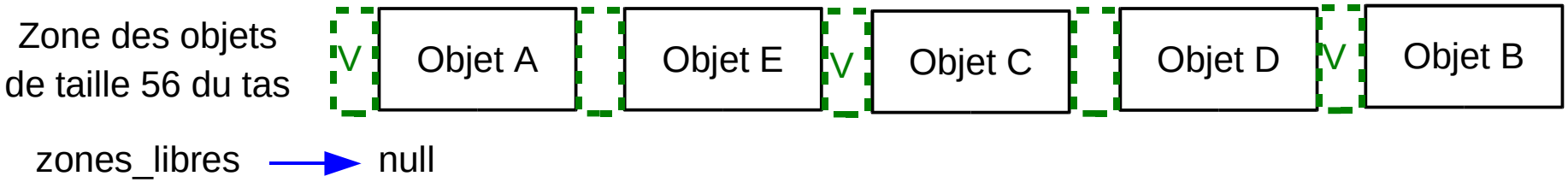


Algorithme de balayage

- On parcourt la liste de toutes les zones mémoires.
- Si marqué, on conserve (on démarque pour le coup suivant),
- Si pas marqué, on intègre la zone dans la liste libre.

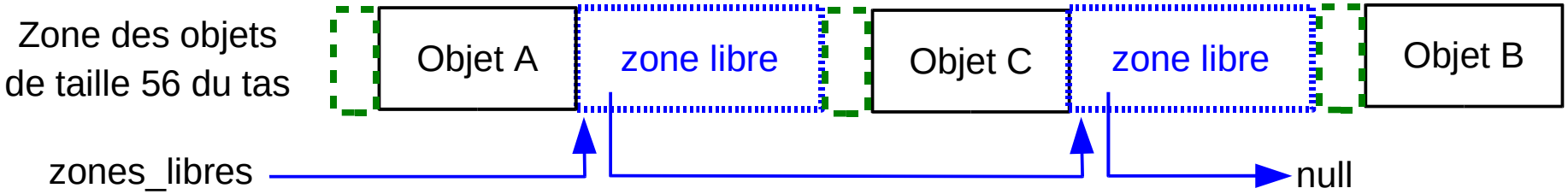
Algorithme de balayage

Après marquage, avant balayage:



Balayage: drapeaux vidés,
zones libres en liste

Après balayage:



Marquage-balayage: bilan

- Pauses longues: marquage + balayage
  - durée du marquage dépend de la taille du graphe d'objets (surtout les vivants)
- Amélioration: incrémental (pauses fractionnées)
- Les cycles ne sont pas un problème
- Fragmentation possible

Copie-compactage

- Principe: parcours du graphe d'objets (comme marquage-balayage) et recopie des vivants dans nouvel espace mémoire (de façon contiguë).
- (cf. page 8 de [JLAP2004] sur ma page web publications)

Copie-compactage: bilan

- Parcours comme marquage
- Recopie coûteuse
- Gestion de « forwarding pointers »
- Double espace mémoire
- Pas de fragmentation

2- Gestion mémoire et temps réel

- TR: +/- rapide, mais prévisible
- Souvent, gestion à la main, bas niveau, placement fixe des structures dans des emplacements fixes
  - flexibilité minimale
  - réutilisabilité très faible
  - prédictibilité totale (capitale en TR dur)

2- Gestion mémoire et temps réel

- Problèmes si utilisation d'algorithmes plus génériques (vus précédemment)
  - Hiérarchie mémoire plus souple amène défauts de page
    - Délai
    - Peu prévisibles
  - Pauses que le GC impose au mutateur
    - Délai
    - Imprévisible

Ramasse-miettes temps réel

- Algorithmes incrémentaux ne suffisent pas en TR
- Développement d'algorithmes TR spécifiques
  - Idée: considérer le GC comme une tâche « comme les autres »
    - Ordonnancé parmi les autres tâches
    - Problème: le faire au bon moment

3- Gestion mémoire et systèmes embarqués

- Systèmes embarqués: omniprésents
- Types de contraintes:
  - Faible taille (code, données, RAM, ROM)
  - Faible puissance de calcul
  - Systèmes embarqués autonomes (tél. PDA, APN...)
    - Faible puissance électrique max (instantanée)
    - Faible autonomie en énergie (cumulée)

Gestion mémoire basse énergie

- Impact de la mémoire sur l'énergie consommée
  - Coût statique (*leakage*): majeure partie
  - Coût dynamique (accès): plus modeste
  - Taille de la mémoire active
  - Type de la mémoire active: RAM, SRAM, etc.
    - Plus est rapide, plus consomme E (sauf disque)

Gestion mémoire basse énergie

- Impact de la mémoire sur l'énergie consommée
  - « éloignement » de la mémoire
    - accès bus très coûteux
  - Gros caches == perfs++ == leakage++ == fréquence-- == perfs--
  - Placement et F d'accès des données importants

Gestion mémoire basse énergie: solutions

- Technologies matérielles (pour mémoire)
  - Conception des cellules SRAM (+ de place, - de fuites): 6 transistors SRAM
  - Partitionnement des bancs mémoire (seul le bloc accédé est activé)
  - *Cache decay*: lignes de cache « mortes » mises en mode basse énergie (*Vdd gating*)
  - Nombreuses autres !

Gestion mémoire basse énergie: solutions

- Idée 1: éteindre ce qui ne sert pas
  - Mémoire en bancs
  - Énergie: grouper (Vitesse: entrelacer)
    - Grouper à l'allocation / selon durée de vie
    - Statiquement ou dynamiquement (ordo. ou GC)
    - Regrouper lors du GC: libérer bancs complets pour les éteindre

Gestion mémoire basse énergie: solutions

- Idée 1: éteindre ce qui ne sert pas
  - Beaucoup de compromis
    - GC quand nouveau banc va être allumé ? Compromis gain/coût...
    - Compression de données ? Compromis gain/coût...
    - Attention au coût (E et T) d'éteindre/rallumer

Gestion mémoire basse énergie: solutions

- Idée 2: limiter le coût E des transferts de données
  - Données très utilisées dans mémoire où accès dynamique peu cher
  - Données peu utilisées dans mémoire où perte statique faible (même si accès cher)
  - Minimiser les transferts (Data Transfert and Storage Exploration): copies des données ?

Gestion mémoire basse énergie: solutions

- Idée 3: diminuer le coût du GC
  - Optimisations en faveur énergie, pas vitesse
    - Marqueurs groupés, pas dans objets
  - Éviter les recopies d'objets
  - ...

Gestion mémoire basse énergie: bilan

- « energy-aware GC »
- Collaboration nécessaire avec le reste du système
  - ordonnancement
  - matériel (API requise)
  - ...
- Pas trivial.